sudo apt install package-manager-session

Agenda

- What is package manager
- History of package management
- Package Format
- Official Repos Structure
- How Package Managers works
- Building package from source

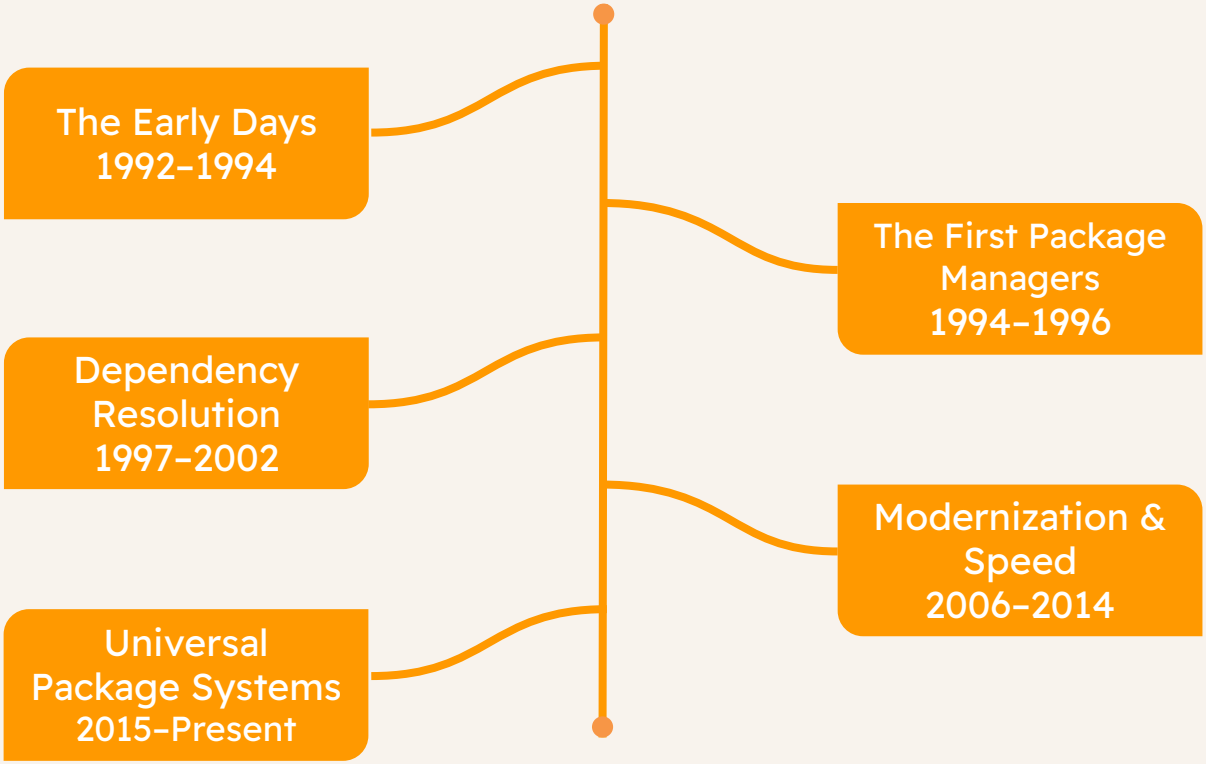
What is package manager ?

Package Manager is a set of tools to download, install, remove, upgrade, configure and manage packages in linux systems.

But Why we need that ?

- Mmmm I think we can go with a tour in history of package management in linux

History of Package Management



Functionalities of Package Manager

Installation

Dependency Resolution

Upgrading

Removing

Querying

Package Format

Package Format

We have multiple packages formats for every package manager to deal with like :

- .deb (Debian packages)
- .rpm (RPM packages for Fedora, openSUSE)
- .pkg.tar.zst (compressed tarballs for Archlinux)
- .xbps (compressed zstd packages for Void Linux)

Debian Packages

Debian packages are ar archives containing three files:

- debian-binary (.deb format version)
- control.tar.gz (metadata)
- data.tar.gz (package files, main scripts)

RPM Packages

RPM package consists of an archive of files and metadata used to install and erase these files and contains the following parts :

- GPG signature
- Header (package metadata)
- Payload (the main package in cpio archive)

Arch Linux Packages

Arch linux packages is a simple tar files (tar balls) compressed in Zstd. These packages has four parts :

- .PKGINFO (metadata)
- .BUILDINFO
- .MTREE (the file structure of the package)
- The main package files

XBPS Packages

XBPS packages are tar archives compressed with Zstd and have these main files:

- files.plist (XML listing all files in the package)
- props.plist (XML for package metadata)
- INSTALL/REMOVE (Optional)
- The main package files

Package Formats

Package	Archive Type	Metadata	Scripts
.deb	ar archive	Control file in control.tar.*	preinst/postinst prerm/postrm
.rpm	(GPG) signature + header + compressed cpio payload	RPM header	%pre, %post, %prerun, %postrun
.pkg.tar.zst	tar + zstd compression	.PKGINFO	Optional .INSTALL in PKGBUILD
.xbps	tar + zstd compression	Props.plist with files.plist for package files	Optional INSTALL and REMOVE

Official Repos Structure

Every distro has its own repositories for the package manager to fetch all the main data from it. If you go to any repository you will find a main files any package manager need to work properly :

- Packages metadata
- The packages itself
- Signatures & Checksums (security concerns)

How Package Managers Works

Install

1. Check local metadata for package info
2. Collect required dependencies, check for conflicts
3. Download package (with dependency packages)
4. Verify integrity (is not corrupted) with checksum
5. Unpack archive, copy package files
6. Run any scripts (if found)
7. Update local package database (pakcage version, file list, dependencies)

Update

1. Connect to configured repositories
2. Download metadata
3. Cache this metadata to use it in install/upgrade next time

Upgrade

1. Fetch latest metadata from repositories
2. Compare installed packages to available versions
3. Detect dependencies for upgrades to prevent any conflict
4. Download the new package with new dependencies
5. Unpack and replace old version
6. Run Scripts (if found)
7. Update local database with new versions and new dependencies

Uninstall

1. Check the local database to know which packages depend on this package
2. If removing is safe, mark the package, dependencies and orphan packages for removal
3. Run any scripts (if found)
4. Delete the packages files
5. Update local database (mark packages as removed)

Search/Query

1. Read local metadata or local installed packages database
2. Show info about packages you search for :
    a. Package name
    b. Version
    c. Dependencies
    d. Installed or not

Important Files

Package Manager	Metadata	Repositories list	Installed Packages
apt	/var/lib/apt/lists	/etc/apt/sources.list	/var/lib/dpkg/status
dnf	/var/cache/dnf/repodata	/etc/yum.repos.d	/var/lib/rpm/*.sqlite
zypper	/var/cache/zypp/raw/	/etc/zypp/repos.d	/var/lib/rpm/*.sqlite
pacman	/var/lib/pacman/sync	/etc/pacman.conf	/var/lib/pacman/local
xbps	/var/db/xbps/<repo-name>	/usr/share/xbps.d	/var/db/xbps/pkgdb-*.plist

HANDS ON TIME

Hands on

1. Download Metadata of packages from your distro’s official repos
2. Found zsh metadata and decompress the file
3. Download zsh package and all dependencies
4. extract the packages (each package in a dir)

NOTE : all downloading will be done by wget

Ex. `wget <url>`

Build from source

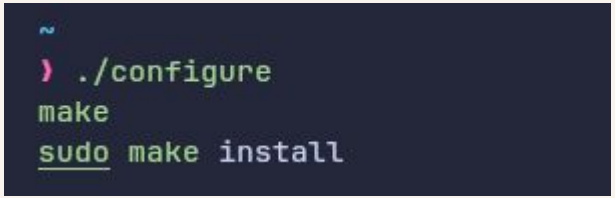
Build From Source

Building from source means compiling and installing software directly from its source code, rather than using pre-built binaries.

This process ensures the software is optimized for your system and up-to-date

Build From Source

1. In Early days (1976), Make invented to build the binaries for the application depending on **MakeFile**. You write the **MakeFile** to manage the dependencies of the source files during the compilation (build) phase.



Build From Source

2. GNU Autotools
    - Autoconf : generate the configure script
    - Automake : generate the Makefile.in from Makefile.am
3. Cmake : lets you define build instructions in CMakeLists.txt to handle cross-platform compilation
4. Ninja : a fast build backend designed to run generated build instructions efficiently

Build From Source

Tool	Category	Purpose
make	Builder	Execute compile commands from MakeFile
autotools	Build system generator	Generate MakeFiles and configure scripts
cmake	Build system generator	Generates build files for multiple builder (make, ninja, etc…)
ninja	Builder	Executes build rules extremely fast
meson	Build system generator	Generates fast ninja files

MakeFile

Makefiles are a list of rules that tell make:

- What to build (targets)
- What they depend on (dependencies)
- How to build them (commands)

MakeFile

We have main things in the Makefile

- Targets
- Dependencies
- Commands
- Variables
- Conditions

Targets

Make deal with targets as actual files. If the file is found in the directory, it will now run this rule and will run it if the file is not there.

Dependencies

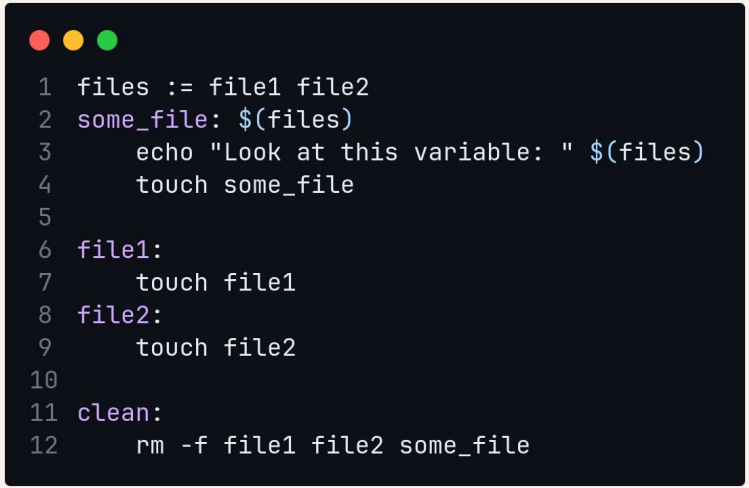
You can specify a dependency for this target to let Make run the dependency target first

Mini Example

```
blah: blah.o
    cc blah.o -o blah

blah.o: blah.c
    cc -c blah.c -o blah.o

blah.c:
    echo "int main() { return 0; }" > blah.c

```

Variables

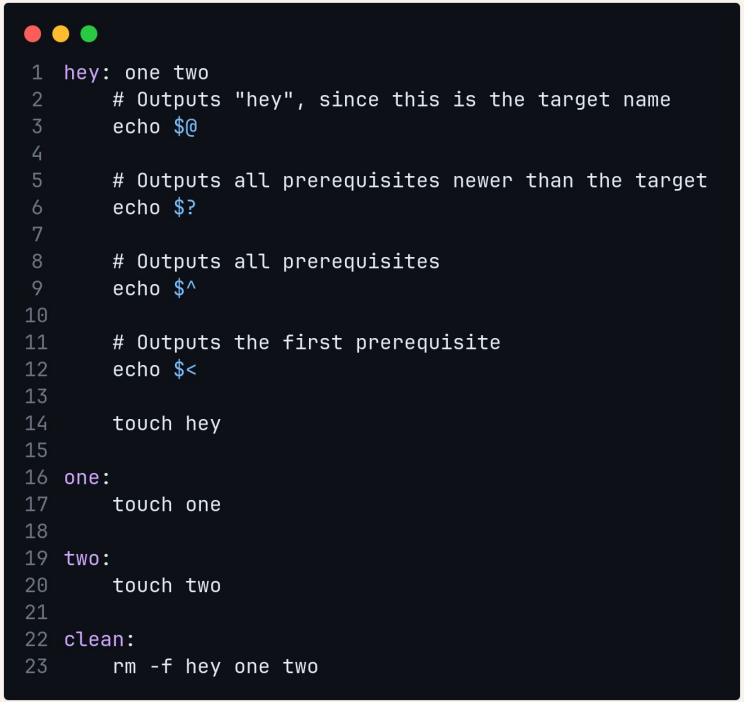
Variables can only be strings. You'll typically want to use := or =



```
files := file1 file2
some_file: $(files)
    echo "Look at this variable: " $(files)
    touch some_file

file1:
    touch file1
file2:
    touch file2

clean:
    rm -f file1 file2 some_file
```

Automatic Variables

Variables that Make can specify by itself



```
hey: one two
    # Outputs "hey", since this is the target name
    echo $@

    # Outputs all prerequisites newer than the target
    echo $?

    # Outputs all prerequisites
    echo $^

    # Outputs the first prerequisite
    echo $<

    touch hey

one:
    touch one

two:
    touch two

clean:
    rm -f hey one two
```

Conditions

You can add conditions to the commands you execute

- ifeq
- ifdef/ifndef

```
emp = $(nullstring)
foo = ok
bar =

equals:
ifeq ($(foo), ok)
    echo "foo equals ok"
else
    echo "nope"
endif

empt:
ifeq ($(strip $(emp)), )
    echo "foo is empty"
endif

def:
ifdef foo
    echo "foo is defined"
endif
ifndef bar
    echo "but bar is not"
endif
```

HANDS ON TIME

Hands on

go to git repo, read the Makefile, build the package and install it on your system

Install :

- zlib-devel
- curl-devel/curl
- openssl-devel
- gettext-devel
- expat-devel
- tcl-devel

THANK YOU